Changements de représentation des données dans le calcul des constructions inductives

Nicolas Magaud, Yves Bertot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Changements de représentation des données dans le calcul des constructions inductives

Nicolas Magaud — Yves Bertot

N° 4039

Octobre 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4039--FR

Changements de représentation des données dans le calcul des constructions inductives

Nicolas Magaud*, Yves Bertot†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 4039 — Octobre 2000 — 29 pages

Résumé : Dans le calcul des constructions inductives, des outils de calcul et de preuve sont associés à chaque type de données concret défini inductivement. Par conséquent, le choix d'une structure de données influence fortement le contenu des preuves. Nous proposons dans ce document une méthode pour passer plus facilement d'une structure de données à une autre, en effectuant des traductions partiellement automatisées des preuves. Un prototype d'outil de traduction a été développé dans le système *Coq* et a été expérimenté sur des traductions de preuves de l'arithmétique de Peano vers l'arithmétique binaire.

Mots-clés : transformation automatique de preuves, calcul des constructions, *Coq*

* Nicolas.Magaud@sophia.inria.fr
† Yves.Bertot@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Changes of Data Type in the Calculus of Inductive Constructions

Abstract: In the calculus of inductive constructions, each time a concrete data type is defined, tools for computing and reasoning about it are also provided. The way proofs are carried out is strongly influenced by the chosen representations of manipulated data. In this paper, we propose a generic approach to ease changes of representation of a datatype by performing partially automatized transformations of proofs. A prototype tool has been developed in the *Coq* proof assistant and was successfully applied to the translation of proofs from Peano's arithmetic to binary arithmetic.

Key-words: automatic proof transformation, calculus of constructions, *Coq*

1 Introduction

Les exigences de qualité de logiciel pour les systèmes embarqués ont conduit à l'introduction de *méthodes formelles* permettant de spécifier de manière très rigoureuse les propriétés que doivent vérifier ces applications. Ces méthodes ont en outre pour but d'établir si un système est conforme ou non à sa spécification.
L'utilisation de systèmes d'aide à la preuve comme *Coq* [HKPM99, Coq99] entre dans ce cadre. De tels outils rendent possible la construction interactive de preuves formelles de théorèmes. A chaque pas de raisonnement effectué par l'utilisateur, la machine vérifie que celui-ci est valide.
Le développement de ces outils a conduit à la formalisation de nombreuses théories mathématiques parmi lesquelles l'arithmétique de Peano, les réels et des notions d'algèbre. Ces théories, qui forment les briques de base de tout développement formel, doivent être maintenues et facilement réutilisables. Il est important de fournir à l'utilisateur les moyens de réorganiser son développement sans tout reprendre à zéro. Des outils d'analyse de dépendances entre théorèmes aidant la maintenance de théories ont été développés récemment [Pon99]. Ils permettent de déterminer quelles parties d'un développement formel peuvent être affectées par la modification d'une définition.

La modélisation d'un problème dans un système d'aide à la preuve conduit à faire des choix d'implantation pour la représentation effective des entités mathématiques considérées. Ces choix influent, non seulement sur la façon de définir des programmes manipulant ces données, mais aussi sur la manière de construire des preuves de propriétés concernant ces objets et ces opérations. La structure d'une preuve dans ce système est en effet fortement liée à celle des types de données concrets utilisés pour modéliser les objets manipulés. On peut être amené à vouloir changer la structure de données décrivant une entité mathématique; par exemple pour avoir un calcul plus efficace.

Le problème que nous nous posons est d'étudier dans quelle mesure, et à quelles conditions, il est possible de réutiliser les preuves déjà construites avant le changement de représentation pour établir les théorèmes relatifs aux opérations définies sur la nouvelle structure de données concrète.

Représentation des preuves Dans un système logique basé sur une théorie des types [Tho91, Dow99], l'isomorphisme de Curry-Howard donne une correspondance entre la logique étudiée et le λ-calcul typé sous-jacent. Les types sont les formules logiques et les λ-termes les preuves. Cet isomorphisme permet d'identifier l'interprétation de la flèche logique $\Rightarrow$ avec la flèche des types fonctionnels $\rightarrow$. De plus, la notion de produit dépendant $\forall x : A.(B\ x)$ est une généralisation de la flèche $A \Rightarrow B$. Lorsque le type $(B\ x)$ ne dépend pas de la valeur de x, on retrouve l'implication logique $A \Rightarrow B$.
Nous montrons, à titre d'exemple, comment construire une preuve de la transitivité de l'implication pour les formules propositionnelles P, Q et R:

$$(R \Rightarrow P) \Rightarrow (P \Rightarrow Q) \Rightarrow R \Rightarrow Q \tag{1}$$

Une preuve de cet énoncé peut être décrite par le λ-terme suivant :

$$\lambda th_1 : (R \Rightarrow P); th_2 : (P \Rightarrow Q); th_3 : R.(th_2\ (th_1\ th_3)) \tag{2}$$

La sémantique de Heyting-Kolmogorov donne aussi une interprétation calculatoire des démonstrations. On peut considérer le terme (2) comme une fonction qui, étant donné les preuves des trois propriétés $(P \Rightarrow Q)$, $(R \Rightarrow P)$ et R, construit une preuve de Q. La première étape est de calculer une preuve de P, au moyen des preuves respectives th_1 et th_3 de $(R \Rightarrow P)$ et de R; cette preuve est donnée par l'application $(th_1\ th_3)$. Il ne reste plus qu'à appliquer th_2 à ce terme de preuve pour obtenir la preuve annoncée de Q, à savoir $(th_2\ (th_1\ th_3))$.

Structure des preuves Un changement de représentation concrète d'une collection d'objets mathématiques consiste non seulement en la modification du type de données concret la décrivant, mais aussi en la construction de nouvelles fonctions de calcul sur ces objets. Si les propriétés P', Q' et R' sont exactement les propriétés P, Q et R adaptées au nouveau type de données, on voit qu'il est possible de réutiliser facilement la preuve (2) pour prouver l'énoncé suivant :

$$(R' \Rightarrow P') \Rightarrow (P' \Rightarrow Q') \Rightarrow R' \Rightarrow Q' \tag{3}$$

Une preuve de cet énoncé est alors :

$$\lambda th'_1 : (R' \Rightarrow P'); th'_2 : (P' \Rightarrow Q'); th'_3 : R'.(th'_2\ (th'_1\ th'_3)) \tag{4}$$

Tout au long de ce document, nous considérons un exemple particulier de changement de type de données concret. Il s'agit du cas des entiers naturels et du passage d'une représentation sous forme des entiers de Peano :

$$nat := \mathsf{O} \mid \mathsf{S}(nat)$$

à une représentation sous forme d'entiers binaires :

$$bin := \mathsf{zero} \mid positif$$

$$positif := \mathsf{one} \mid 2 * p\ (p > \mathsf{zero}) \mid 2 * p + 1\ (p > \mathsf{zero})$$

Cette transformation conduit à un changement radical de la structure des preuves par induction. En effet, le nombre de constructeurs passe de deux à quatre et il n'y a aucune correspondance entre les pas d'induction dans les deux types de données considérés. On ne peut pas réutiliser facilement la preuve d'un théorème dans une représentation pour établir le théorème équivalent dans l'autre représentation.

Pour être capable de réutiliser les preuves établies dans la première représentation, il faut disposer d'un principe d'induction non structurelle sur la nouvelle représentation; ce principe doit permettre de conserver la structure du raisonnement de départ.

Problèmes liés à la convertibilité Une autre difficulté apparaît s'il existe une relation d'équivalence par réduction sur les termes. Cette équivalence entre deux termes peut disparaître lors de leurs traductions vers une nouvelle représentation. Si dans notre calcul, P_A et P_B sont équivalents, le terme

$$\lambda th_1 : (P_A \Rightarrow Q); th_2 : (R \Rightarrow P_B); th_3 : R.(th_1\ (th_2\ th_3)) \tag{5}$$

est une preuve de l'énoncé suivant :

$$(P_A \Rightarrow Q) \Rightarrow (R \Rightarrow P_B) \Rightarrow R \Rightarrow Q \tag{6}$$

Supposons que les termes P'_A et P'_B, traductions de P_A et P_B dans la nouvelle structure de données, ne soient plus convertibles; il est nécessaire de modifier la structure du terme de preuve (5) et de remplacer l'équivalence implicite entre P_A et P_B par l'application d'un théorème $th_{conv} : P'_B \Rightarrow P'_A$, qu'il faudra établir séparément. Une preuve de l'énoncé traduit de (6), à savoir

$$(P'_A \Rightarrow Q') \Rightarrow (R' \Rightarrow P'_B) \Rightarrow R' \Rightarrow Q'$$

est donc le terme suivant :

$$\lambda th'_1 : (P'_A \Rightarrow Q'); th'_2 : (R' \Rightarrow P'_B); th'_3 : R'.(th'_1\ (th_{conv}\ (th'_2\ th'_3)))$$

Objectifs Nous présentons dans ce document des résultats pratiques visant à faciliter le passage d'une structure de données à une autre dans le développement formel d'une théorie mathématique. Nous proposons de fournir :

- un moyen de construire des principes de récurrence non structurelle sur des types de données concrets;
- la possibilité de construire systématiquement les énoncés des théorèmes d'égalité nécessaires pour compenser les problèmes liés à la convertibilité;
- un algorithme traduisant automatiquement les preuves d'une représentation vers une autre en utilisant les outils introduits dans les deux points précédents.

Nous étudions principalement les cas des entiers naturels et de leurs codages par les entiers de Peano et les entiers binaires. Nous décrivons toutefois une méthode générale applicable à d'autres changements de structures de données concrètes.

Nous commençons par présenter un certain nombre de travaux antérieurs utiles à notre étude (Section 2). Nous donnons ensuite un aperçu des caractéristiques du système *Coq* et en particulier de la manipulation des types inductifs (Section 3). Dans un troisième temps, nous montrons quelles opérations nous devons effectuer pour préparer la traduction automatique de preuves suivant un changement de structure de données (Section 4). Nous décrivons ensuite notre algorithme de traduction de preuves et présentons les résultats obtenus (Section 5). Nous concluons finalement sur le travail réalisé avant de présenter les perspectives ouvertes par cette expérience.

2 Contexte de l'étude

Nous décrivons brièvement les travaux antérieurs reliés à notre sujet d'étude et montrons en quoi ils nous ont influencé. Le premier porte sur la formalisation de l'algèbre universelle en théorie des types [Cap99]. Nous résumons ensuite les différents travaux traitant de transformations automatiques de preuves et leurs applications [Mad89, And94, Ric95, MW99, Den00].

2.1 Présentation universelle des données

Venanzio Capretta décrit dans [Cap99] une formalisation de la notion d'algèbre universelle en théorie des types. L'intérêt d'un tel développement est qu'il permet d'avoir un cadre générique pour la spécification de structures de données. En utilisant ce cadre, on peut facilement représenter les théories du premier ordre décrivant les entiers naturels et leurs opérations classiques sous forme d'une algèbre de termes (avec les symboles O, S, plus...). Il est ensuite envisageable d'exprimer dans ce même langage formel ce qu'est un théorème sur les entiers naturels (par un ensemble de règles de construction par exemple). A partir de là, il devient possible de construire deux interprétations de ces théorèmes dans les entiers de Peano et dans l'arithmétique binaire. On peut ensuite envisager d'établir un principe qui montre que tout théorème sur les entiers de Peano a un équivalent sur les entiers binaires.

Cette approche présente l'avantage d'être très formelle et générale, mais l'inconvénient d'être difficilement utilisable en pratique. Elle se rapproche des techniques de réflexion [Bou97] qui consistent à exprimer dans un système des propriétés du système lui-même. Ici, on raisonnerait sur les objets du système (les théorèmes sur les entiers naturels) en les modélisant à l'intérieur même du système sous forme d'un type de données concret.

Nous n'avons pas retenu cette technique car le principe à établir demande un effort considérable pour décrire le cadre logique dans lui-même. De plus, le théorème d'incomplétude de Gödel montre que cet objectif ne peut être atteint que partiellement.

2.2 Transformations automatiques de preuves

Nous nous intéressons aux techniques de transformation automatique de preuves afin de réutiliser les preuves construites avec la représentation initiale des données pour établir les preuves correspondantes dans la nouvelle représentation. De nombreux travaux ont été effectués sur des sujets similaires aussi bien dans le cadre de la synthèse de programmes [Mad89, And94, Ric95] que dans celui des preuves par analogie [MW99] visant à réutiliser l'expérience acquise dans les preuves précédentes. Ewen Denney [Den00] a par ailleurs proposé un mécanisme de traduction de preuves construites avec le système HOL vers un format de preuves acceptable par le système *Coq*.

2.2.1 Transformation de preuves et synthèse de programmes

Le paradigme *proofs-as-programs* consiste à extraire d'une preuve constructive de l'énoncé :

$$\forall x.P(x) \;\Rightarrow\; \exists y \mid Q(x,y)$$

un programme fonctionnel calculant y à partir de x. Plusieurs études ont été réalisées concernant la transformation ou l'optimisation de programmes par modification de la preuve à partir de laquelle ils sont synthétisés.
Madden [Mad89] propose de transformer automatiquement des programmes par la transformation automatique de la preuve de leur spécification. Il montre, en particulier, comment des transformations au niveau de la preuve permettent la spécialisation du programme extrait. Penny Anderson [And94] présente un moyen d'encoder les transformations de preuve au sein du système Elf. Elle propose d'optimiser un programme en modifiant directement la structure de la preuve sous-jacente plutôt que d'utiliser les techniques standards (purement syntaxiques) d'optimisation. Elle montre, entre autres, comment utiliser cette méthode pour transformer une définition de fonction récursive en une définition récursive-terminale.

Julian Richardson [Ric95] utilise les transformations de preuve pour changer les structures de données mises en jeu dans la preuve et le programme qui en est extrait. Cela permet en disposant de structures de données sur lesquelles les opérations de base sont plus efficaces d'améliorer les performances des programmes extraits. Considérons par exemple un changement de représentation des entiers naturels : le passage des entiers de Peano aux entiers binaires. On se donne la définition des entiers de Peano et de l'addition sur ce type de données, la définition des entiers binaires et des fonctions de passage d'une représentation à l'autre. A partir de ces éléments, il est possible de construire automatiquement une preuve de la spécification

$$\forall x,y \in \mathsf{bin}, \exists z \in \mathsf{bin} \mid nat(z) = nat(x) + nat(y)$$

où *nat* est une fonction de traduction des entiers binaires vers les entiers de Peano. Le travail présenté dans [Ric95] permet d'obtenir une preuve de cette propriété à partir de laquelle il est possible de synthétiser un programme raisonnablement efficace de calcul de l'addition sur les entiers binaires. Ce résultat pourrait être utilisé dans notre travail pour synthétiser les opérations de base sur les entiers binaires. Cependant, il ne permet pas d'obtenir l'algorithme le plus efficace de calcul de la somme de deux entiers binaires. Nous avons donc préféré considérer que la définition des opérations de base sur un nouveau type concret fait partie des données initiales du problème.

2.2.2 Preuves par analogie

Un autre sujet de recherche intéressant est l'utilisation du raisonnement par analogie dans les assistants de preuve. Il est en effet fréquent que des preuves mathématiques informelles contiennent des formules comme *"On démontre P de la même façon que l'on a établi Q"*.

On peut distinguer deux étapes dans la preuve par analogie :

- choisir, dans une base de données, un théorème Q sur lequel on va s'appuyer pour construire la démonstration du nouveau théorème P; celui-ci est parfois fourni par le rédacteur de la preuve ;
- adapter la preuve de Q pour que celle-ci soit une preuve de P; c'est une tâche qui incombe au lecteur de la preuve.

Dans le cadre des systèmes d'aide à la démonstration de théorèmes, des travaux concernant l'automatisation de ces deux aspects du raisonnement par analogie ont été effectués. Régis Curien propose dans [Cur95] une méthode pour rechercher dans une base de théorèmes un énoncé dont la preuve pourrait être réutilisée pour établir le nouvel énoncé considéré. Cette recherche se base sur la notion de filtrage d'ordre 2 sur l'énoncé du théorème dont on cherche à établir une preuve. Supposons que l'on cherche à établir une preuve de $\forall n,m \in \mathsf{nat}\ (\mathsf{mult}\ n\ m) = (\mathsf{mult}\ m\ n)$, on construit un motif dans lequel on peut avoir des variables qui représentent des objets du premier ordre et des fonctions sur ces objets (d'où le terme "2^e ordre"). Dans l'exemple précédent, le motif construit est le suivant $(f\ x\ y) = (f\ y\ x)$ avec f un symbole de fonction et x et y des variables du premier ordre. Le système cherche ensuite dans la base de données un énoncé qui est une instance de ce motif, par exemple $\forall n,m \in \mathsf{nat}\ (\mathsf{plus}\ n\ m) = (\mathsf{plus}\ m\ n)$.

Erica Melis et Jon Whittle proposent dans [MW99] une technique pour construire cette nouvelle preuve à partir du modèle. On commence par établir une correspondance entre les objets mis en jeu dans chacun des deux théorèmes, puis on rejoue en parallèle la construction des preuves de l'ancien et du nouveau théorème tant que le but (le théorème courant à établir) dans le théorème cible vérifie les critères autorisant l'application du même pas de preuve que dans l'original. Si ces conditions ne sont pas vérifiées, on recourt à des heuristiques classiques résolvant le but, ou bien on imagine un lemme permettant l'établissement de la propriété recherchée. On tentera de prouver ce lemme séparément. L'invention d'un lemme s'accompagne de l'utilisation d'un *disprover* qui, par la recherche de contre-exemples, vérifie si le lemme proposé n'est pas trivialement faux.

Ces idées ont été implémentées avec succès au sein du démonstrateur de théorèmes CLAM. Ce système est basé sur des tactiques et des *proof-plans* (qui peuvent être considérés comme des super-tactiques permettant une large automatisation des preuves). Les calculs de recherche de preuve effectués par ces tactiques peuvent être arbitrairement compliqués. Dans notre approche, nous travaillons sur une description exacte de la preuve indépendamment de la façon dont elle a été construite. Le système *Coq* donne un moyen plus précis de contrôler la traduction des preuves. En effet, en cas d'échec de l'application d'une règle dans un *proof-plan*, il est difficile d'en déterminer les raisons et de résoudre le problème. Dans le cas d'un terme de preuve, les étapes de raisonnement considérées sont élémentaires et en cas d'échec, on peut plus facilement trouver quelle autre règle appliquer pour pouvoir poursuivre la traduction de la preuve.

2.2.3 Traduction d'un formalisme vers un autre

Ewen Denney [Den00] propose un mécanisme de traduction permettant de générer des scripts de preuve *Coq* à partir d'un enregistrement du déroulement d'une preuve dans le système HOL. HOL (acronyme de *Higher-Order Logic*, logique d'ordre supérieur) est un système de preuves dans la tradition de LCF. Seuls les énoncés des théorèmes sont représentés dans un λ-calcul simplement typé. En particulier, il n'y a pas de représentation des preuves sous forme de λ-termes, celles-ci sont représentées sous forme d'une succession d'applications de règles d'inférence. Il n'existe par ailleurs pas de notion de convertibilité comme c'est le cas dans *Coq*.

Un mécanisme de traduction tel que celui décrit dans [Den00] permet la réutilisation des preuves développées en HOL pour construire de nouvelles théories dans *Coq*; on évite ainsi de reformaliser les mêmes notions mathématiques dans des systèmes différents.
Le problème de la coopération entre les systèmes de preuves a déjà été étudié dans le cas des systèmes HOL et Nuprl [FH97]. Felty et Howe ont réalisé une formalisation hybride d'une preuve de normalisation de la logique combinatoire SK simplement typée en utilisant à la fois des bibliothèques de théorèmes dans les deux systèmes. De tels travaux ont pour objectif de montrer qu'il n'est pas nécessaire de redéfinir une théorie si elle est déjà formalisée dans un autre système. Elle peut, en effet, soit être utilisée directement, soit traduite systématiquement dans le formalisme désiré.

Lors d'une traduction comme celle proposée par Denney, on est amené à exprimer des notions primitives dans le formalisme de départ par des théorèmes dédiés "les imitant" dans le formalisme d'arrivée. C'est le cas pour les opérateurs de réécriture au niveau de l'équivalence propositionnelle dans HOL. Les étapes de réécriture sont remplacées par l'application d'un théorème *Coq* simulant un pas de réécriture.
Cette technique de remplacement de manipulations primitives de termes par l'application de théorèmes sera réutilisée dans notre travail.

3 Présentation du système Coq

Nous commençons par décrire rapidement le système d'aide à la preuve *Coq*, puis nous présentons les types inductifs dans ce système. Ceux-ci permettent la définition de types de données concrets et sont un élément central de notre étude.

3.1 Caractéristiques générales

Le système *Coq* est l'implantation d'un formalisme logique d'ordre supérieur : le Calcul des Constructions Inductives [Coq99, Chap. 4]. Il s'agit d'un λ-calcul typé avec types dépendants [Dow99], permettant en outre la définition de constructions inductives [PM93].
Coq propose un mode interactif de développement de preuve par chaînage arrière. On dispose d'un certain nombre de tactiques, applicables au but courant si certaines préconditions sont vérifiées. Elles permettent d'effectuer les pas élémentaires de preuve.

Dans ce calcul, les preuves sont des termes du λ-calcul comme les autres. L'isomorphisme de Curry-Howard permet d'interpréter les types du calcul comme des formules logiques et les λ-termes comme des preuves. L'objet dénotant la preuve d'un théorème est conservé; il est donc possible de vérifier une preuve par une simple opération de vérification de type.

3.2 Les types inductifs dans Coq

3.2.1 Définitions

L'extension inductive du calcul des constructions et son implantation *Coq* disposent d'une notion de types inductifs primitifs. Pour chaque type inductif, le système propose deux constructions primitives de filtrage (Cases) et de définition par point-fixe (Fixpoint); les schémas de récurrence sont définis à partir de celles-ci.

Ces types inductifs permettent de définir des collections d'objets mathématiques de la même manière que l'on définit des types de données concrets en ML. Par exemple, les entiers naturels *à la Peano* sont définis de la manière suivante dans *Coq* :

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

Cela signifie que l'on définit un objet nat, de sorte Set, dont les constructeurs sont O (zéro) et S (la fonction successeur $x \mapsto x+1$).
La définition d'un type inductif dans *Coq* provoque la construction *automatique* d'un *principe d'induction structurelle* et d'un opérateur de récursion permettant de définir des fonctions récursives. Le principe d'induction sur les entiers de Peano prend la forme suivante :

```
nat_ind
  : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

qui correspond exactement au principe de récurrence classique sur les entiers :

$$P\,(0) \;\wedge\; (\forall n \in \mathsf{nat}\,.\; P\,(n) \Rightarrow P\,(n+1)) \;\Rightarrow\; \forall n \in \mathsf{nat}\,.\; P\,(n) \tag{7}$$

3.2.2 Convertibilité

Il existe dans le calcul des constructions inductives une règle de convertibilité qui identifie certains termes entre eux. Cette règle transcrit la notion de $\beta\delta\iota$-équivalence, où la β-réduction correspond à l'application de fonctions, la δ-réduction au remplacement de noms de constantes par leurs définitions effectives et la ι-réduction définit les règles de réduction associées aux opérateurs Cases et Fixpoint.

Considérons par exemple la définition dans *Coq* de la fonction factorielle :

```
Fixpoint fact [n:nat] : nat :=
    Cases n of O => (S O)
             | (S p) => (mult (fact p) (S p))
end.
```

Les règles de ι-réduction correspondant à cette définition sont les suivantes :

$$(\mathsf{fact}\ \mathsf{O}) \to_\iota (\mathsf{S}\ \mathsf{O}) \qquad (\mathsf{fact}\ (\mathsf{S}\ p)) \to_\iota (\mathsf{mult}\ (\mathsf{fact}\ p)\ (\mathsf{S}\ p)))$$

Cela signifie que le système ne fait aucune distinction (autre que syntaxique) entre les deux termes (fact O) et (S O) . On dit que ces termes sont intentionnellement égaux, ou convertibles. Ils peuvent être utilisés indifféremment l'un à la place de l'autre.

Conséquences de la convertibilité La preuve de l'égalité (fact (3))=(6) dans un système comme *Coq* est immédiate. Les deux termes considérés (fact (3)) et (6) se réduisent à la même forme normale par les règles de réduction du calcul; ils sont donc intentionnellement égaux. Une telle preuve est dite implicite, puisque le raisonnement est en fait remplacé par du calcul et que le terme de preuve construit exprime simplement que (6)=(6), masquant que le membre gauche de l'équation a fait l'objet d'une réduction. L'application des règles de convertibilité ne construit aucun terme de preuve et se contente de modifier l'aspect syntaxique de l'énoncé à établir.
De la même manière, l'énoncé

$$\forall n \in \mathsf{nat} \quad (\mathsf{fact}\ (\mathsf{S}\ n)) = (\mathsf{mult}\ (\mathsf{fact}\ n)\ (\mathsf{S}\ n))$$

se prouve de façon immédiate par le calcul; en effet la ι-réduction du terme de gauche conduit au terme de droite. Il est important de savoir que cette preuve serait difficile si S n'était pas un constructeur, mais une simple fonction.

Une présentation très complète de l'utilisation des types inductifs dans *Coq* est faite dans [Gim98] et les fondements théoriques sont présentés dans [PM93].

4 Codage d'une structure dans une autre

Le but de cette partie est de présenter les éléments à mettre en place afin de pouvoir faire une traduction automatique des termes de preuves lors d'un changement de représentation d'une notion mathématique. Nous commençons par étudier le passage de principes d'induction d'une représentation dans l'autre. Une fois ces principes disponibles, nous étudions comment remplacer les règles de convertibilité (en particulier la ι-conversion) par des étapes de réécriture utilisant des théorèmes d'égalité.

4.1 Représentation des données et principes d'induction

Nous étudions comment il est possible, à partir du codage d'une structure de données dans une autre, de définir des principes de récurrence non structurelle sur un type de données concret. Le but de ceci est de permettre de conserver la structure des preuves au moment d'un changement de représentation. Nous traitons plus particulièrement le cas où les types de données concrets mis en jeu sont isomorphes.

Considérons le cas d'un changement de représentation pour les entiers naturels. Nous reprenons la notion d'entiers binaires telle qu'elle est définie dans l'implantation de la tactique Omega [Coq99, Chap. 16] de *Coq*.

La définition des entiers binaires dans *Coq* est donné par les lignes suivantes :

```
Inductive positif : Set :=
      one: positif              (* 1 *)
    | pI: positif ->positif     (* 2x+1, x>0 *)
    | p0: positif ->positif.    (* 2x, x>0 *)

Inductive bin : Set := zero : bin | pos : positif -> bin.
```

Il est possible, et plus naturel, de définir les entiers binaires comme des listes de booléens. Cette représentation, bien que plus simple que celle adoptée, présente cependant l'inconvénient de ne pas donner une écriture unique de chaque entier naturel. Par exemple, 0100 et 100 sont deux représentations différentes du nombre quatre.

A la suite de ces définitions, les principes d'induction structurelle positif_ind et bin_ind sont automatiquement engendrés. On obtient finalement un principe d'induction sur les entiers binaires très différent du principe d'induction (7) dérivé de la définition des entiers de Peano. Ce nouveau principe d'induction comporte quatre cas mutuellement exclusifs : 0, 1, $2p+1$, $2p$ avec $(p > 0)$.

$$P\ (0)\ \wedge\ P\ (1)\ \wedge\ (\forall p \in \mathsf{positif}\ .\ P\ (p) \Rightarrow P\ (2p+1))\ \wedge\ (\forall p \in \mathsf{positif}\ .\ P\ (p) \Rightarrow P\ (2p))$$
$$\Rightarrow\ \forall n \in \mathsf{bin}\ .\ P\ (n)$$

D'un point de vue ensembliste, les types bin et nat sont isomorphes (il est possible de construire une application bijective de l'un dans l'autre) ; mais les principes d'induction et opérateurs de récursion qui leur sont associés les distinguent fortement. De plus, l'opération de calcul du successeur d'un entier s'exprime de manière notablement plus compliquée dans l'arithmétique binaire que dans l'arithmétique de Peano.

```
Definition S' : bin -> bin :=
   [n:bin] Cases n of zero => (pos one)
            | (pos u) => (pos (aux_S' u))
           end.

Fixpoint aux_S' [n:positif] : positif :=
   Cases n of one (* 1 *)            => (p0 one)        (* 2*1 *)
            | (p0 t) (* 2*t t>0 *)   => (pI t)          (* 2*t+1 *)
            | (pI t) (* 2*t+1 t>0 *) => (p0 (aux_S' t)) (* 2*(t+1) *)
   end.
```

Grâce à l'ordre supérieur, nous allons énoncer et prouver un nouveau principe d'induction sur les entiers binaires:

$$P\ (\mathsf{zero})\ \wedge\ (\forall n \in \mathsf{bin}\ .\ P\ (n) \Rightarrow P\ (\mathsf{S'}\ n))\ \Rightarrow\ \forall n \in \mathsf{bin}\ .\ P\ (n) \qquad (8)$$

Ce nouveau principe suivra la structure des entiers de Peano et permettra ainsi de conserver la structure des preuves établies par récurrence dans l'arithmétique de Peano. Le développement complet de la construction de ce nouveau principe d'induction en *Coq* est donné à l'annexe B.1.

Procédé de construction

1. Nous commençons par construire deux fonctions

$$\mathsf{Nconvert} : \mathsf{bin} \rightarrow \mathsf{nat} \qquad \mathsf{anti\_Nconvert} : \mathsf{nat} \rightarrow \mathsf{bin}$$

   permettant le passage d'une représentation binaire à une représentation *à la Peano* des entiers naturels et vice-versa.
2. Dans un deuxième temps, nous établissons à l'aide des définitions précédentes et de quelques propriétés élémentaires sur l'addition des entiers binaires l'égalité suivante :

$$\forall a \in \mathsf{bin} \quad \mathsf{anti\_Nconvert}(\mathsf{Nconvert}(a)) = a$$

3. A partir de cette égalité, nous substituons dans la conclusion de l'énoncé du principe d'induction recherché (8) la variable n par $\mathsf{anti\_Nconvert}(\mathsf{Nconvert}(n))$.
4. Il est alors possible d'appliquer le principe d'induction sur les entiers de Peano nat_ind avec comme prédicat $Q = P\ o\ \mathsf{anti\_Nconvert}$ sur le terme $(\mathsf{Nconvert}(n))$ de type nat. Les buts résultant de cette application sont facilement établis en utilisant les hypothèses du principe d'induction attendu.

Nous disposons maintenant d'un principe d'induction non structurelle (8) sur les entiers binaires. Les preuves procédant par induction dans l'arithmétique de Peano peuvent donc être réutilisées pour établir les théorèmes de base de l'arithmétique sur les entiers binaires. Il nous reste tout de même à fournir les éléments permettant de compenser la disparition de la convertibilité entre certains termes lors d'un changement de structures de données.

4.2 Représentation des fonctions

Un changement de représentation est souvent motivé par une volonté de disposer d'une structure de données plus efficace pour les calculs. Les opérations de base sur les données doivent donc être redéfinies. Les fonctions sur les entiers binaires doivent, pour être efficaces, être définies de façon structurelle sur leurs arguments. En fait, leur définition ne fait que suivre la technique de calcul manuel sur les entiers en base 2. Dans l'arithmétique de Peano, la définition de l'addition est la suivante :

```
Fixpoint plus[n:nat] : nat -> nat :=
   Cases n of O => [m:nat]m
    | (S p) => [m:nat](S (plus p m))
   end.
```

Elle conduit au fait que les termes (plus (S n) m) et (S (plus n m)) sont convertibles, puisque le premier se réduit en le second par simple ι-réduction. Dans la formalisation utilisant les entiers binaires, l'addition Bplus est définie par récursion structurelle sur les entiers binaires (voir annexe B.2) et les termes (Bplus (S' n) m) et (S' (Bplus n m)) ne sont pas convertibles. Malheureusement, les preuves bâties dans l'arithmétique de Peano utilisent de manière implicite ces propriétés de convertibilité. Si l'on veut pouvoir traduire les preuves sur l'arithmétique de Peano vers des preuves sur les entiers binaires, il est indispensable de produire des théorèmes d'égalité que l'on utilisera pour simuler par des réécritures la convertibilité implicitement utilisée dans les preuves originales.

Génération automatique des théorèmes d'égalité Il est possible, à partir de la définition de l'addition sur les entiers de Peano de construire automatiquement les énoncés des théorèmes d'égalité à établir sur l'addition des entiers dans l'arithmétique binaire.

$$\forall m \in \mathsf{bin} \quad (\mathsf{Bplus\ zero}\ m) = m \tag{9}$$

$$\forall p,m \in \mathsf{bin} \quad (\mathsf{Bplus}\ (\mathsf{S'}\ p)\ m) = (\mathsf{S'}\ (\mathsf{Bplus}\ p\ m) \tag{10}$$

Il suffit pour cela d'exprimer pour chaque branche de l'opérateur (Cases) que l'application de la fonction (ici plus) au motif de filtrage est égale au membre droit de cette branche et de traduire l'énoncé ainsi obtenu vers l'opération d'addition sur les entiers binaires. De façon à pouvoir raisonner facilement sur de telles égalités, on prend soin d'appliquer la fonction considérée à tous ses arguments. De cette façon, on aura à établir et à manipuler des égalités sur des types non fonctionnels. Nous n'avons pas étudié la question de savoir s'il sera avantageux de conserver des types fonctionnels et de prouver les théorèmes (Bplus zero) = λm : bin.m et $\forall p \in$ bin,(Bplus (S' p)) = λm : bin.(S' (Bplus p m)).

Construire automatiquement les preuves de (9) et (10) apparaît comme un problème plus difficile et reste à étudier. Pour chaque nouvelle opération sur les entiers binaires, on doit établir ces théorèmes d'égalité. Dans l'outil que nous proposons, ceux-ci sont stockés dans une base de données nommée simplification, et seront utilisés pour faire de la réécriture automatique. Ces égalités sont orientées de gauche à droite suivant ainsi le sens de la ι-réduction mise en jeu dans l'arithmétique de Peano.

4.3 Mise en correspondance de deux théories

On relie chaque objet intervenant dans les preuves utilisant les entiers de Peano à un objet jouant le même rôle dans le développement relatif aux entiers binaires. Le tableau 1 présente les composantes de base de cet isomorphisme. On peut remarquer que le constructeur O de l'arithmétique de Peano est traduit en le constructeur zero de l'arithmétique binaire, alors que le constructeur S est traduit en la fonction S' qui n'est plus un constructeur dans l'arithmétique binaire. L'isomorphisme ainsi construit doit être étendu à chaque fois qu'un nouveau théorème T_{nat} est traduit en un théorème T_{bin}.

Implantation Dans l'outil que nous proposons, au niveau du système *Coq*, l'isomorphisme est géré dans une table de hachage. On associe à chaque donnée dans l'arithmétique de Peano son équivalent dans l'arithmétique binaire. On dispose de deux commandes pour modifier et observer cet isomorphisme :

- "AddIso t_{nat} t_{bin} ." ajoute une liaison dans la table ;
- "CheckIso t_{nat}." permet de connaître (s'il en existe un) le terme de l'arithmétique binaire associé au terme de l'arithmétique de Peano t_{nat}.

Type de données	nat	bin
Constructeurs dans	O	zero
l'arithmétique de Peano	S	S'
Opérations de base	plus	Bplus
	mult	Bmult
Principes d'induction	nat_ind	new_bin_ind
Théorèmes	plus_n_O	plus_n_O_bin

TAB. 1 – *Données de base fournies par l'isomorphisme*

5 Traduction systématique des termes de preuve

Nous disposons maintenant d'un principe d'induction non structurelle sur les entiers binaires qui imite le principe d'induction sur les entiers de Peano. De plus, nous avons montré comment gérer le problème de la convertibilité lors d'un changement de structure de données. Nous pouvons donc passer à la description du procédé de traduction adopté pour obtenir des théorèmes sur les entiers binaires à partir des théorèmes sur les entiers de Peano.

5.1 Script ou terme de preuve

La preuve d'un théorème est représentée dans le système *Coq* par un λ-terme; il existe cependant une autre manière de stocker les preuves. Le script de preuve est une transcription de haut niveau des interactions effectuées avec le système. Il est constitué de la liste des tactiques appliquées et de leurs paramètres. Les étapes de raisonnement mettant en jeu la convertibilité sont parfois précisées (tactique Simpl). Cependant, le script de preuve ne contient pas nécessairement d'information relative à l'utilisation de la convertibilité. Le script donne la façon dont la preuve a été construite alors que le terme n'est qu'un témoin que le théorème est prouvé. Le terme de preuve fournit une description de plus bas niveau de la preuve, en la décrivant uniquement sous forme d'abstractions et d'applications de termes du calcul. L'utilisateur a toujours la possibilité dans un script de preuve de fournir directement un terme de preuve au moyen de la tactique Exact. Pour savoir traduire les scripts de preuve,

il est donc nécessaire de savoir traduire les termes de preuve. Par ailleurs, le script de preuve n'apporte pas d'information véritablement utile par rapport au terme de preuve; il est donc apparu plus simple de travailler directement au niveau des termes de preuve.

5.2 Exemple de terme de preuve à traduire

Dans la suite de ce document, nous considérons comme exemple de théorème à traduire plus_n_O :

$$\forall n \in \mathsf{nat} \quad n = (\mathsf{plus}\ n\ \mathsf{O}) \tag{11}$$

Le λ-terme suivant est une preuve de l'énoncé précédent :

$$\begin{array}{ll} \lambda n : \mathsf{nat}.(\mathsf{nat\_ind} & \lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \\ & (\mathsf{refl\_equal\ nat\ O}) \\ & \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})). \\ & \quad (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)\ n) \end{array}$$

Le terme (refl_equal nat O) est une preuve de O = O. Par convertibilité, c'est aussi une preuve de O = (plus O O). De même,

$$\lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)$$

est une preuve de

$$\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{S}\ (\mathsf{plus}\ n0\ \mathsf{O}))$$

C'est aussi une preuve de $\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{plus}\ (\mathsf{S}\ n0)\ \mathsf{O})$ qui est le type attendu pour le troisième argument du principe de récurrence nat_ind.

5.3 Terme de preuve traduit structurellement

La première étape de la traduction d'un terme de preuve consiste en la traduction syntaxique du terme en suivant l'isomorphisme présenté dans la section 4.3. On parcourt simplement le terme en substituant les entités ayant une traduction par l'isomorphisme par leur équivalent dans la théorie résultat. A ce stade de la transformation, le terme n'est probablement pas typable, à cause de problèmes de convertibilité.

A l'issue de cette étape, notre exemple devient :

$$\begin{array}{ll} \lambda n : \mathsf{bin}.(\mathsf{new\_bin\_ind} & \lambda n0 : \mathsf{bin}.n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}) \\ & (\mathsf{refl\_equal\ bin\ zero}) \\ & \lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})). \\ & \quad (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)\ n) \end{array} \tag{12}$$

new_bin_ind est le principe d'induction (8) sur les entiers binaires suivant la structure des entiers de Peano.
Il faut maintenant transformer le terme ci-dessus (qui n'est pas typable en l'état) en une preuve de la traduction de l'énoncé (11), à savoir :

$$\forall n \in \mathsf{bin} \quad n = (\mathsf{Bplus}\ n\ \mathsf{zero}) \tag{13}$$

5.4 Double calcul de type

Lors de la première phase de traduction, nous avons ignoré tous les problèmes relatifs à la convertibilité. Certaines étapes de preuve étaient effectuées de manière implicite en utilisant les règles de réduction du calcul. Il nous faut maintenant déterminer, dans le terme de preuve, les positions où un problème de typage est apparu lors de la transformation. Nous définissons pour cela un double calcul de type, suivant le même principe que les travaux de Yann Coscoy [Cos96] pour la traduction en langage naturel des preuves dans *Coq*.

5.4.1 Notions de type attendu et de type proposé

L'algorithme de traduction que nous présentons dans la suite utilise les notions de type attendu et de type proposé. Le type attendu T_A détermine l'information imposée par le contexte qui va recevoir un terme t. Le type proposé T_P est le type effectif du terme t synthétisé par le système.
Dans le cas d'une application $(t_1\ t_2)$, telle que le système de type de *Coq* infère les types suivants :

$$t_1 : T \to U \qquad \& \qquad t_2 : T'$$

Le type attendu pour t_2 est le type argument T du type fonctionnel $T \to U$, alors que le type proposé est le type inféré T' par le système pour t_2. Pour que le terme $(t_1\ t_2)$ soit typable, il faut que les types attendu et proposé soient convertibles. Si l'on interprète les types comme des formules logiques et les termes comme des preuves, le type attendu T correspond à une *obligation de preuve*. Il indique l'énoncé dont on doit fournir une preuve. Le type proposé T' est un *constat de preuve*. Il donne l'énoncé effectivement prouvé par le terme considéré (dans ce cas t_2). Pour que le raisonnement soit correct, il faut que l'implication $T' \Rightarrow T$ soit prouvable.

5.4.2 Algorithme utilisé

La structure de données manipulée est celle des termes. Les termes du calcul que nous considérons ici sont les abstractions (λ-abstractions et types produits), les applications et les variables. D'autres termes existent (les constantes, l'opérateur de filtrage Cases...); ils ne sont pas présentés car l'algorithme ne fait que les retourner sans aucune modification.

```
terme := abstr (variable, terme, terme) | prod (variable, terme, terme)
         appl (terme, liste_de_termes) | variables
```

A partir d'un terme t et d'un type T_A (le type attendu pour t), l'algorithme retourne un terme t' de type T_A et une liste d'énoncés de théorèmes à prouver pour que le terme t' soit bien typable et de type T_A. Ces théorèmes sont ceux destinés à remplacer la convertibilité entre les termes par de la réécriture sur les théorèmes d'égalité introduits à la section 4.2. Nous présentons informellement le fonctionnement de cet algorithme.
Initialement, le contexte (composé de liaisons variable $\rightarrow$ type) est vide. L'algorithme travaille par analyse structurelle du terme de preuve t en cours de traduction (par exemple le terme (12)).

Abstraction Dans le cas où t est une abstraction $\lambda x : A.t'$, on réduit le type attendu T_A pour t sous forme d'un type produit $\forall x : A'.(T'\ x)$. On utilise pour cela l'algorithme de mise en forme normale de tête faible. On ajoute dans le contexte le fait que x est de type A, puis on rappelle l'algorithme sur le terme t' avec le type attendu $(T'\ x)$. Si A est un type non-dépendant (par exemple bin ou bin $\rightarrow$ bin), A' sera nécessairement le même type (puisque le terme initial dans l'arithmétique de Peano était bien typé). Si A est un type dépendant, il ne peut s'agir que d'un type dénotant une formule logique (par exemple $n = (\mathsf{Bplus}\ n\ \mathsf{zero})$) ; en effet, on ne considère jamais, dans notre étude, de termes mettant en jeu des types dépendants sur des objets sur lesquels on calcule. Dans ce cas, le type $(T'\ x)$ ne dépend pas de x puisque, en vertu du principe de *proof-irrelevance*, on ne peut pas construire une propriété dépendante d'une preuve particulière.
En supposant que le résultat retourné par cet appel récursif est un terme t'', l'algorithme reconstruit une abstraction $\lambda x : A.t''$ et compare alors le type T_P de ce terme au type attendu $T_A \equiv \forall x : A'.(T'\ x)$. Si ces types sont convertibles, il n'y a plus rien à faire et le résultat $\lambda x : A.t''$ est retourné. Dans le cas contraire, il faut utiliser une conjecture établissant l'implication entre le type proposé T_P et le type attendu T_A; le système calcule un nouveau nom pour cet énoncé, par exemple H. Cette propriété devra être établie séparément pour terminer la traduction. Dans ce cas, le terme retourné est $(\mathsf{H}\ (\lambda x : A.t''))$.

Application Dans le cas où t est une application $(\mathsf{hd}\ t_1 \ldots t_n)$, le type T_{hd} de la tête hd de l'application est synthétisé par le système. Il n'y a pas de notion de type attendu pour la tête de l'application. A partir de ce type T_{hd}, on peut calculer le type attendu T_{A1} pour le premier argument (par la mise en forme normale de tête faible du type T_{hd}). On rappelle l'algorithme sur ce premier argument t_1. Trois résultats sont possibles:

- t_1 **n'est pas typable.** Dans ce cas, l'algorithme est rappelé récursivement sur les données t_1 et T_{A1} et retourne un terme t'_1 convenable.
- t_1 **est typable, mais de type non convertible avec** T_{A1}**.** Supposons que le type inféré par le système pour t_1 soit T_{P1}, on introduit alors une conjecture $\mathsf{H} : T_{P1} \Rightarrow T_{A1}$, et $(\mathsf{H}\ t_1)$ donne un terme de type T_{A1}.
- t_1 **est typable et convertible avec le type attendu** T_{A1}**.**
  Il n'y a plus rien à faire; le terme t_1 convient.

Une fois le premier terme de l'application traduit et bien typé (notons ce terme t'_1), on type le terme $(\mathsf{hd}\ t'_1)$ pour obtenir le type attendu T_{A2} pour le deuxième terme t_2.

On itère le procédé jusqu'à avoir appliqué tous les termes à la tête de l'application en cours de traduction. Il faut ensuite typer la nouvelle application obtenue ($\mathsf{hd}\ t'_1 \ldots t'_n$) pour vérifier si le type proposé T_P pour ce terme est compatible avec le type attendu T_A. Si oui, on retourne ($\mathsf{hd}\ t'_1 \ldots t'_n$); dans le cas contraire, on retourne ($\mathsf{H}\ (\mathsf{hd}\ t'_1 \ldots t'_n)$) où H est l'énoncé $T_P \Rightarrow T_A$ que l'on devra établir séparément.

Variables Les variables ne sont pas transformées. Il est cependant important de savoir que le système *Coq* dispose de deux représentations des variables liées : les variables nommées et les indices de De Bruijn. Lors de l'introduction d'une variable dans le contexte (au moment de la déstructuration d'une abstraction ou d'un produit), il est nécessaire de nommer la variable et de remplacer les indices de De Bruijn la dénotant dans le terme par son nom.

Les constantes, les types inductifs et leurs constructeurs ont déjà été traduits dans la phase de traduction structurelle du terme de preuve. Ils ne sont donc pas modifiés. La traduction de l'opérateur de filtrage Cases est prise en compte par l'algorithme dans le cas des entiers naturels.

Raisonnement par cas avec l'opérateur Cases Le raisonnement par cas sur une structure peut être utilisé au moyen de la tactique Case. Au niveau du terme de preuve, son utilisation conduit à l'apparition de l'opérateur de filtrage primitif Cases. Lors du changement de structure de données, on ne peut plus utiliser cet opérateur pour faire du raisonnement par cas suivant l'ancien type de données. Cet opérateur ne permet en effet que de manipuler des objets de manière structurelle. On est donc amené à appliquer un théorème particulier pour reproduire ce raisonnement par cas :

$$P(\mathsf{zero}) \Rightarrow (\forall n : \mathsf{bin}\ P(\mathsf{S'}\ n)) \Rightarrow \forall n : \mathsf{bin}\ P(n)$$

Ce traitement est spécifique à l'étude du cas de la traduction de preuves de l'arithmétique de Peano vers l'arithmétique binaire. Sa généralisation demande de pouvoir retrouver les noms de théorèmes à utiliser lors de la traduction d'un opérateur de filtrage à partir du type sur lequel on fait une étude par cas. Ce travail parait facilement réalisable. Il n'a néanmoins pas fait l'objet d'une implantation.

Les conjectures utilisées pour faire coïncider type attendu et type proposé pour un terme peuvent contenir des variables libres (présentes dans le contexte de la preuve). Cependant, afin de pouvoir les établir effectivement (voir section 5.5) en utilisant les tactiques du système *Coq*, il est nécessaire de les clore par rapport à leurs variables libres.

5.4.3 Comportement de l'algorithme sur l'exemple

Dans notre exemple, seul le terme correspondant à la traduction du pas d'induction (troisième argument de new_bin_ind) nécessite l'utilisation d'une conjecture H_conv pour faire coïncider les types proposé et attendu. Le terme traduit structurellement

$$\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}))\ (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)\ n)$$

a le type

$$\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{S'}\ (\mathsf{Bplus}\ n0\ \mathsf{zero}))$$

qui n'est pas convertible avec le type attendu

$$\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{Bplus}\ (\mathsf{S'}\ n0)\ \mathsf{zero})$$

L'algorithme engendre comme obligation de preuve l'énoncé suivant :

$$\begin{array}{l}\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{S'}\ (\mathsf{Bplus}\ n0\ \mathsf{zero})) \\ \Rightarrow \forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{Bplus}\ (\mathsf{S'}\ n0)\ \mathsf{zero})\end{array}$$

Si H_conv est une preuve de cet énoncé, le terme traduit structurellement précédent devient

$$(\mathsf{H\_conv}\ (\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}))\ (\mathsf{f\_equal}\ \mathsf{bin}\ \mathsf{bin}\ \mathsf{S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)))$$

Le terme de preuve global

$$\begin{array}{lll}\lambda n : \mathsf{bin}.(\mathsf{new\_bin\_ind} & \lambda n0 : \mathsf{bin}.n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}) & \\ & (\mathsf{refl\_equal}\ \mathsf{bin}\ \mathsf{zero}) & \\ & (\mathsf{H\_conv}\ (\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})). & (14) \\ & \quad (\mathsf{f\_equal}\ \mathsf{bin}\ \mathsf{bin}\ \mathsf{S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)))\ n)\end{array}$$

est alors typable et son type est $\forall n : \mathsf{bin}\ n = (Bplus\ n\ \mathsf{zero})$. Il ne reste plus qu'à établir l'obligation de preuve pour que la traduction soit complète.

5.5 Résolution des obligations de preuve engendrées

A chaque fois qu'un problème de convertibilité est détecté lors de la phase précédente, le système engendre un énoncé de théorème lui permettant d'obtenir le bon type. Ces énoncés sont ajoutés artificiellement au contexte pour terminer la traduction du terme de preuve; ils doivent cependant être établis formellement afin que la traduction soit complète. Dans nos expériences, ces conjectures ont toujours pu être établies au moyen de la tactique

Repeat (Intro ; AutoRewrite [simplification]) ; Prolog [] 3.

Cette tactique introduit une à une les hypothèses et tente à chaque fois de réécrire le but en utilisant l'ensemble des théorèmes d'égalité présent dans la base de données simplification construite à la section 4.2. La tactique Prolog permet de prouver automatiquement des théorèmes en utilisant les hypothèses locales et est capable de gérer des variables existentielles; cela permet d'appliquer des hypothèses dont certaines variables apparaissent dans les prémisses, mais pas dans la conclusion (typiquement des hypothèses de transitivité d'une relation binaire).

La tactique de réécriture automatique AutoRewrite dispose comme règles de réécriture des égalités orientées de gauche à droite simulant la convertibilité présente dans la représentation initiale des données. Ces règles simulent la ι-réduction sur les termes traduits. Comme une propriété de normalisation forte du calcul des constructions inductives pour les règles β, δ, ι a été établie [Coq99, Chap. 4], on a la garantie que la réécriture automatique faite par la tactique AutoRewrite terminera.

5.6 Expérimentation et limitations

Nous avons implémenté cet algorithme en ML et intégré ce code au système *Coq*. Les expériences que nous avons menées ont montré que cette commande était tout à fait utilisable en pratique. Pour pouvoir traduire l'ensemble des théorèmes de la bibliothèque d'arithmétique mettant en jeu l'addition et la multiplication, nous avons dû faire quelques traitements spécifiques à la représentation des entiers sous forme binaire. Les preuves de théorèmes mettant en jeu les propriétés d'injectivité ou de non-confusion des constructeurs des entiers de Peano ont reçu un traitement *ad-hoc*.

Injectivité et séparabilité des constructeurs Les constructeurs d'un type de données ont des propriétés particulières par rapport aux fonctions habituelles. Ils sont injectifs et libres (c'est-à-dire que deux termes égaux ne peuvent pas être obtenus à l'aide de constructeurs différents). Les preuves de théorèmes dépendants de ces propriétés utilisent l'opérateur de filtrage sur le type de données concret considéré. Cependant celui-ci n'est utilisable que pour des raisonnements suivant la structure du terme. Il est donc nécessaire de recourir à un théorème simulant l'opérateur de filtrage. Cette technique nécessite l'établissement de règles de réduction relatives au principe d'induction new_bin_ind. Il faut être capable d'établir l'égalité correspondant à la réduction suivante :

$$(\mathsf{new\_bin\_ind}\ P\ \mathsf{h\_base}\ \mathsf{h\_rec}\ \mathsf{zero}) \rightarrow \mathsf{h\_base}$$

Une autre égalité semblable doit être établie pour le cas (S' p). Nous n'avons pas établi ces égalités qui semblent difficiles. La solution, plus pratique, que nous avons choisi est de construire des théorèmes bin_injection et bin_discriminate dont les énoncés sont les suivants :

$$\forall n,m \in \mathsf{bin}\ (\mathsf{S'}\ n) = (\mathsf{S'}\ m) \Rightarrow n = m \qquad \forall n \in \mathsf{bin}\ \neg\mathsf{zero} = (\mathsf{S'}\ n)$$

On les utilise à la place de la construction primitive Cases dans les preuves. Pour faire cela, il est nécessaire de savoir reconnaître dans les preuves les fragments utilisant l'injectivité et la non-confusion des constructeurs. Ces fragments de preuve étant construits automatiquement, c'est assez facile. Il suffit de chercher dans les termes de preuve les motifs suivants : (LET ? ? ? (eq_ind ...) ...) pour la discrimination des constructeurs et (LET ? ? ? (f_equal ...)...) pour l'injectivité.

Cas des prédicats inductifs La définition d'un prédicat inductif représentant une propriété logique conduit automatiquement à la construction d'un principe de raisonnement sur

sa structure. Dans le cas de la relation d'ordre $\leq$ sur les entiers binaires, nous définissons un ordre sous une forme facilement calculable en suivant la structure des entiers binaires LE, puis nous reprenons et adaptons la définition de l'ordre le_bin sur les entiers de Peano aux entiers binaires. A partir de ces données, il suffit d'établir deux théorèmes le_bin_vers_LE et LE_vers_le_bin établissant que ces deux relations sont équivalentes pour pouvoir continuer de raisonner sur les entiers binaires en utilisant le principe de raisonnement associé à la définition de l'ordre $\leq$ sur les entiers de Peano. Les preuves mettant en jeu la relation d'ordre $\leq$ sur les entiers de Peano peuvent donc elles aussi être facilement traduites. Cette partie n'a pas été implantée, mais testée à la main sur quelques exemples.

Traduction de la bibliothèque de théorèmes de base de Coq Nous avons pu traduire automatiquement toute la bibliothèque de théorèmes de base sur l'addition et la multiplication dans l'arithmétique de Peano vers l'arithmétique binaire grâce à cette commande.

6 Conclusions et perspectives

6.1 Travail réalisé

Ce travail a permis d'étudier les méthodes à mettre en œuvre pour pouvoir réutiliser plus facilement, lors d'un changement de représentation d'une notion mathématique, les bibliothèques de théorèmes existantes. Pour cela, nous avons proposé des outils pour raisonner de manière non structurelle sur un type de données concret. Les moyens de définir facilement un principe d'induction non structurelle ont été étudiés. Disposer d'un tel principe permet la traduction structurelle des preuves. Cependant, la notion de convertibilité entre termes présente dans le système *Coq* complique cette traduction, il est nécessaire d'établir explicitement certaines égalités entre termes dans le développement utilisant le nouveau type de données. Ces égalités étaient déjà présentes dans l'ancienne représentation mais sous forme calculatoire; elles étaient implicites au niveau des termes de preuve. Une fois un principe d'induction non structurelle défini, et après avoir proposé une méthode de construction des lemmes d'égalité relatifs à la ι-conversion perdue, nous avons développé un traducteur automatique de preuves. Nous l'avons testé avec succès sur le cas des entiers naturels. Nous avons traduit toute la bibliothèque de théorèmes sur les entiers de Peano utilisant l'addition et la multiplication vers les théorèmes équivalents dans l'arithmétique binaire.

6.2 Perspectives

L'intérêt des outils de traduction proposés est de faciliter les changements de structures de données dans les théories formelles développées avec un outil tel que *Coq*. Les perspectives de recherche ouvertes par cette étude sont nombreuses. On peut tout d'abord regarder comment automatiser la construction des principes d'induction et des théorèmes d'égalité préalables à la traduction automatique des preuves. Il parait possible d'automatiser la gé-

nération de l'énoncé du principe d'induction; il est cependant moins facile de savoir si le prouver automatiquement dans certains cas est envisageable.

L'approche présentée est relativement générale et ne dépend que peu du fait que l'on a travaillé sur les entiers naturels (codés sous forme d'entiers de Peano ou d'entiers binaires). On peut étudier comment cette approche se généralise à d'autres changements de structures de données. Le codage choisi pour les entiers binaires n'est pas le codage habituel sous forme de liste de booléens. Il serait intéressant de voir comment traduire des théorèmes de l'arithmétique de Peano vers des théorèmes utilisant cette structure de données. On sera alors confronté à un codage non injectif, ce qui posera de nouvelles questions comme le choix de représentants canoniques ou l'utilisation d'ensembles quotients.

On peut aussi envisager d'utiliser des méthodes semblables dans le cadre de la spécification formelle de machines virtuelles (Java par exemple). En effet, la mémoire de telles machines est souvent modélisée de façon simple par des listes. Une fois que des propriétés de ces machines ont été établies formellement avec cette représentation de la mémoire, on peut être tenté d'améliorer l'implantation en utilisant des arbres binaires à la place des listes. Cependant, dans l'état actuel des choses, cette modification nécessite de refaire toutes les preuves. Il serait intéressant d'étudier dans quelles mesures les travaux présentés dans ce document peuvent s'appliquer dans ce cas.

Références

[And94] Penny Anderson. Representing proof transformations for program optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814, pages 575–589. LNAI, Springer-Verlag, 1994.

[Bou97] S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[Cap99] Venanzio Capretta. Universal algebra in type theory. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.

[Coq99] The Coq Team. The Coq Proof Assistant Reference Manual – Version V6.3. Technical report, INRIA, July 1999. Version révisée distribuée avec Coq.

[Cos96] Y. Coscoy. A natural language explanation for formal proofs. In C. Retoré, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Liguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.

[Cur95] Régis Curien. *Outils pour la preuve par analogie.* PhD thesis, Université Henri Poincaré Nancy I, INRIA-Lorraine, 1995.

[Den00] Ewen Denney. A prototype proof translator from hol to coq. In *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs'2000*, LNCS. Springer, 2000.

[Dow99] Gilles Dowek. Théories des types, 1999. disponible à http://pauillac.inria.fr/~dowek/cours.html.

[FH97] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using nuprl and hol. In *Fourteenth International Conference on Automated Deduction*, volume 1249 of *LNCS*. Springer, 1997.

[Gim98] Eduardo Giménez. A tutorial on recursive types in coq. Rapport technique 221, INRIA, May 1998. disponible à http://coq.inria.fr/doc-eng.html.

[HKPM99] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.2.4. Rapport technique, INRIA, January 1999. Version révisée distribuée avec Coq.

[Mad89] Peter Madden. The specialization and transformation of constructive existence proofs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 131–148. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 416.

[MW99] E. Melis and J. Whittle. Analogy in inductive theorem proving. In *Journal of Automated Reasoning*, volume 22, pages 117–147, 1999.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[Pon99] Olivier Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs.* PhD thesis, Conservatoire National des Arts et Métiers, 1999.

[Ric95] Julian Richardson. Automating Changes of Data Type in Functional Programs. In *Proceedings of KBSE-95, Boston USA, 12-15 November 1995.* IEEE Computer Society, 1995. Extended version available from Edinburgh as DAI Research Paper 767.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison - Wesley, 1991. ISBN 0-201-41667-0.

A Exemple de session

Nous présentons un exemple d'utilisation de notre commande de traduction de preuves. La propriété considérée est l'associativité à gauche de l'addition plus_assoc_l.

Après avoir chargé les bibliothèques contenant les propriétés décrites à la section 4, nous demandons au système de nous montrer le terme de preuve de l'associativité à gauche de l'addition sur les entiers de Peano. Nous appelons ensuite notre tactique de traduction sur ce terme. Le système *Coq* nous retourne le théorème intermédiaire plus_assoc_l_rr1 nécessaire ainsi que le terme prouvant le théorème traduit plus_assoc_l' dans l'arithmétique binaire.

```
rem:~/Translate$ coqtop -init-file COQRC.v
```

```
Welcome to Coq V6.3.1 (December 1999)
[Reinterning Induction...done]
[Reinterning Definitions...done]
[Reinterning Arith...done]
[Reinterning Le...done]
[Reinterning Lt...done]
[Reinterning Plus...done]
[Reinterning Gt...done]
[Reinterning Minus...done]
[Reinterning Mult...done]
[Reinterning Between...done]
[Reinterning Wf_nat...done]
[Reinterning MyWf...done]
[Loading ML file g_natsyntax.cmo ...done]
[Reinterning Regles_add...done]
[Reinterning Isomorphism...done]
[Loading ML file isomorphism.cmo ...done]
[Reinterning Translate...done]
[Loading ML file translate.cmo ...done]

Coq < Print plus_assoc_l.
plus_assoc_l =
[n,m,p:nat]
 (nat_ind [n0:nat](plus n0 (plus m p))=(plus (plus n0 m) p)
   (refl_equal nat (plus m p))
   [n0:nat; H:((plus n0 (plus m p))=(plus (plus n0 m) p))]
    (f_equal nat nat S (plus n0 (plus m p)) (plus (plus n0 m) p) H) n)
     : (n,m,p:nat)(plus n (plus m p))=(plus (plus n m) p)

Coq < Time Translate plus_assoc_l.
------------------------------------------------------
Lemma plus_assoc_l_rr1 : (p,m:bin)
 ((n0:bin)
   (Bplus n0 (Bplus m p))=(Bplus (Bplus n0 m) p)
   ->(S' (Bplus n0 (Bplus m p)))=(S' (Bplus (Bplus n0 m) p)))
 ->(n:bin)
    (Bplus n (Bplus m p))=(Bplus (Bplus n m) p)
    ->(Bplus (S' n) (Bplus m p))=(Bplus (Bplus (S' n) m) p)
.
Repeat (Intro ; AutoRewrite [simplification]); Prolog [] 3.
Qed.

Theorem plus_assoc_l' : (n,m,p:bin)(Bplus n (Bplus m p))=(Bplus (Bplus n m) p)
.
Exact [n,m,p:bin]
 (new_bin_ind [n0:bin](Bplus n0 (Bplus m p))=(Bplus (Bplus n0 m) p)
   (refl_equal bin (Bplus m p))
   (plus_assoc_l_rr1 p m
     [n0:bin; H:((Bplus n0 (Bplus m p))=(Bplus (Bplus n0 m) p))]
      (f_equal bin bin S' (Bplus n0 (Bplus m p)) (Bplus (Bplus n0 m) p)
        H)) n)
.
Qed.
```

```
AddIso plus_assoc_l plus_assoc_l'.
------------------------------------------------------------
Finished transaction in 0 secs (0.15u,0s)

Coq <
```

B Extraits du développement réalisé en Coq

B.1 Preuve du nouveau principe d'induction

Définition des fonctions de traduction entre nat et bin

```
Fixpoint aux_convert [p : positif] : nat :=
 Cases p of   one => (S O)
             | (pI t) => (S (mult (aux_convert t) (2)))
             | (pO t) => (mult (aux_convert t) (2))
 end.

Definition Nconvert : bin -> nat :=
   [n : bin]
       Cases n of   zero => O
                  | (pos p) => (aux_convert p)
       end.

Fixpoint anti_Nconvert [n : nat] : bin :=
 <bin> Cases n of
     O => zero
    | (S p) => (S' (anti_Nconvert p))
 end.
```

Preuve de l'invariance point par point de bin par anti_Nconvert o Nconvert

```
Lemma invariant_bin: (v : bin)  (anti_Nconvert (Nconvert v)) = v.
Intros v; Elim v.
Simpl; Auto.
Intros p; Elim p.
Simpl; Auto.
Intros pO H'; Simpl.
Rewrite mult_sym; Simpl.
Rewrite (plus_sym (aux_convert pO) O); Simpl.
Rewrite anti_Nconvert_plus.
Simpl in H'.
Rewrite H'.
Simpl.
Rewrite <- aux_Bplus_double.
Simpl; Auto.
Intros pO H'; Simpl.
Rewrite mult_sym; Simpl.
Rewrite (plus_sym (aux_convert pO) O); Simpl.
Rewrite anti_Nconvert_plus.
Simpl in H'.
```

```
Rewrite H'.
Simpl.
Rewrite <- aux_Bplus_double; Auto.
Qed.
```

Preuve du nouveau principe d'induction

```
Lemma new_bin_ind: (P : bin -> Prop) (P zero) -> ((n : bin) (P n) -> (P (S' n)))
 -> (l : bin)  (P l).
Intros P H H0 l; Try Assumption.
Rewrite <- (invariant_bin l).
Elim (Nconvert l).
Simpl; Exact H.
Simpl.
Intros n H1; Try Assumption.
Apply H0.
Exact H1.
Qed.
```

B.2 Définition de l'addition sur les entiers binaires

```
Fixpoint
   aux_Bplus [x, y:positif]: positif :=
      Cases x of
          one => Cases y of
                     one => (p0 one)
                   | (pI t) => (p0 (aux_S' t))
                   | (p0 t) => (pI t)
                 end
        | (pI u) => Cases y of
                        one => (p0 (aux_S' u))
                      | (pI t) => (p0 (aux_plus_carry' u t))
                      | (p0 t) => (pI (aux_Bplus u t))
                    end
        | (p0 u) => Cases y of
                        one => (pI u)
                      | (pI t) => (pI (aux_Bplus u t))
                      | (p0 t) => (p0 (aux_Bplus u t))
                    end
      end
   with
   aux_plus_carry' [x, y:positif]: positif :=
      Cases x of
          one => Cases y of
                     one => (pI one)
                   | (pI t) => (pI (aux_S' t))
                   | (p0 t) => (p0 (aux_S' t))
                 end
        | (pI u) => Cases y of
                        one => (pI (aux_S' u))
                      | (pI t) => (pI (aux_plus_carry' u t))
                      | (p0 t) => (p0 (aux_plus_carry' u t))
```

```
                    end
        | (p0 u) => Cases y of
                        one => (p0 (aux_S' u))
                      | (pI t) => (p0 (aux_plus_carry' u t))
                      | (p0 t) => (pI (aux_Bplus u t))
                    end
      end.

Definition Bplus: bin -> bin ->bin :=
   [n, m:bin]
      Cases n of
          zero => m
        | (pos p) => Cases m of
                         zero => (pos p)
                       | (pos q) => (pos (aux_Bplus p q))
                     end
      end.
```

Table des matières

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399